© 2019 Aravind Sagar

AQUEDUCT: TASK-BASED ENTRY POINTS IN ANDROID APPS

BY

ARAVIND SAGAR

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2019

Urbana, Illinois

Adviser:

Assistant Professor Ranjitha Kumar

ABSTRACT

Modern smartphones offer voice assistants to ease a variety of tasks. However, the actions that can be performed by current voice assistants are limited – a predefined set of built in actions like checking the weather, and a few hooks that can be built into third-party applications. To extend assistant actions to third-party applications, the onus is on the application developers to manually add support for voice assistant integration. To improve the link between voice assistants and third-party apps, we built Aqueduct, a data driven task-based app search and task entry point discovery system for Android. We search over app UI data augmented with semantic annotations to find applications and screens within those applications that can accomplish a given task. Furthermore, Aqueduct can leverage the package name and the activity name of the discovered screen to automatically navigate users to that screen. A user study was conducted to compile a set of common smartphone tasks and evaluate the effectiveness of Aqueduct, which showed that it is effective at finding task-based entry points for a wide range of tasks.

Aqueduct is also useful for augmenting search in application repositories, finding starting points for execution for task-automation systems, and even generating deep link suggestions for applications.

To my parents, Resmy R P and Suhruth S D, for their omnipresent love and support.

ACKNOWLEDGMENTS

Thank you to Professor Ranjitha Kumar for all the opportunities and guidance that she has given me. Thank you to Ph.D. student Deniz Arsan and graduate student Oliver Melvin for their contributions to the project. Thank you to the entire Data Driven Design Group for their feedback and support.

Thank you to my parents, Suhruth and Resmy, and my sister Daya, for their love and support. Thank you to my entire family and friends for their encouragement and blessings.

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION

Voice controlled intelligent assistants like Google Assistant, Amazon's Alexa and Apple's Siri have risen in popularity recently. Quality of speech recognition has improved due to massive online processing, and spoken mode of communication is often more natural and faster than typing for many users [1]. A study commissioned by Google found out that 55% of the U.S. teens use voice search every day and that 89% of teens and 85% of adults agree that voice search is going to be "very common" in the future [2].

Despite the increasing trend of using voice assistants to perform tasks in smartphones, the set of actions supported by these assistants are often limited. Most assistants natively support common simple tasks such as checking the weather or setting an alarm, and recent efforts like "Actions on Google" provide a mechanism for application developers to provide hooks that can be invoked by a voice assistant [3]. However, in all of these cases, the actions supported by voice assistants are predefined and limited. Furthermore, it requires non-trivial effort from the developers to add support for voice assistant integration with their applications. In most cases where the voice assistant does not know how to handle a given query, their response is to perform a web search and show users the results, which isn't very useful. A more useful way to handle unknown tasks would be to find third-party apps capable of performing the specified task and redirecting users to one of them. This is what Aqueduct aims to achieve.

Aqueduct is task-based app search and task entry point discovery system for Android. It matches user tasks (e.g., *"send money"*) to mobile apps, as well as a screen within each of those apps which can act as an entry point for that task. In addition, Aqueduct can also use the package name and the activity name of the discovered screen to automatically navigate users to that screen directly, instead of just opening the home screen of an app.

Aqueduct finds the task-based entry points by using app UI data augmented with semantic annotations. A task often specifies an action such as "shop" or "send," and an object of that action such as "clothing" or "money." UI elements such as icons and buttons often represent actions, while the object of these actions can be inferred from surrounding text fields. Therefore, Aqueduct combines icon, button, and text data from UI screens with corresponding Play Store metadata to find tasks-based entry points in Android apps.

Mining app UI data is an integral part of the system, and to achieve this, we improved upon prior app UI mining infrastructure to make it more scalable and reliable, and use previously recorded app UI data (the *Rico* dataset, comprising of 66k UI screens from 9.3k Android applications) in addition to the newly collected data (9.5k screens from 176 appli-

cations) [4] [5].

We conducted a 24-person study to evaluate the effectiveness of Aqueduct. Using the system, participants were able to find an application for 64% of their tasks, and were satisfied with the discovered entry points in 95% of these cases. The study also suggested that Aqueduct could be used to improve application search in mobile app repositories like the Google Play Store, since many app listings contains advertising images designed to look appealing to the user, instead of actual screenshots. Aqueduct on the other hand can provide real screenshots from apps, tailored to the search query.

CHAPTER 2: RELATED WORK

2.1 APPLICATION DISCOVERY

Most mobile application discovery systems focus on presenting useful applications to users based on their download history and usage patterns [6], or their context information (like location and time) [7]. Some work has also been done on finding relevant applications for a given query, and the approach usually involves searching through UI data collected from applications through automatic or manual exploration [8].

Prior work has also focused on tagging mobile applications with keywords indicating key features and concepts in the application [9]. This approach finds existing tagged applications similar to a given untagged application, and mines the text data on those applications to discover relevant tags for the given application. This has the potential to provide better results than searching through all the text in an application.

2.2 VOICE ASSISTANTS AND THIRD-PARTY APPS

Deep links into applications provide a starting point for executing tasks in Android. Currently, the onus is on the application developer to manually expose deep links and provide actions supported by their application. Android allows application developers to manually enable deep links into their applications using intent filters [10]. Using this system, developers can specify the URI and/or the MIME type of the data that their application is able to handle. This type of linking is targeted towards navigating to content inside applications, and hence the actions available for inclusion for developers is limited. To enable a wider set of actions for use with voice assistants, Google recently started developing "Actions on Google", which provide a more diverse set of actions [3]. The rollout of these actions is being done on a per-category basis, and currently includes actions related to food ordering, finance, fitness, and ride sharing. However, it is up to the developers to manually add support for these predefined actions. Application developers can also develop custom conversational actions on the Google assistant, but this requires a certain level of query processing on their side [11].

Researchers have also attempted to help developers add deep links into their applications. One study introduced a library that allows developers to add dynamic links and achieve higher coverage of their applications through static and dynamic application analysis [12]. Other studies have also proposed RESTful style mobile application models to improve service

discovery and content search within applications [13].

None of these methods allow automatically connecting voice assistants to functionality within applications without some sort of developer effort.

2.3 AUTOMATED TASK EXECUTION

A lot of prior work on programmatically navigating Android applications is in the context of automated testing. The widely used Monkey framework exercises the UI by sending a random stream of touches to a device [14]. Researchers have found techniques to improve the input sequences in order to maximize test coverage [15]. Prior work has also leveraged machine learning to automate test generation [16]. However, these techniques generally do not apply to automating tasks in real-world scenarios, since they are aimed at simplifying UI testing and maximizing coverage of tests.

End user task automation systems presently available generally takes the form of a service that allows users to create Event-Condition-Action (ECA) rules [17]. This triggers the execution of a predefined action when a particular event is observed. While many systems provide predefined ECA rules, they are usually fully customizable by the user. While this works well for simple tasks, they usually support only simple, predefined actions.

A more flexible task execution system called *SUGILITE* has also been proposed [18]. This is a programming-by-demonstration system, where the user manually completes the task once, which is recorded and used to automate the task in the future. To do this, *SUGILITE* uses Android's accessibility framework to get apps' UI structure, and manipulate the apps' user interface.

Google is working on a yet-to-be-released simple task automation system which works on mobile websites through Google assistant [19], without any changes in the target applications. This system supports limited predefined tasks (car rentals and movie ticket bookings are the first planned items).

2.4 COMMON MOBILE TASKS

To facilitate discovery of actions supported by applications, a representative set of common tasks that mobile users perform would be very useful. However, publicly available studies that analyze smartphone usage are at the application level, not task level [20, 21, 22, 23, 24]. The largest of these studies was done with *AppSensor* in 2011, but the focus was contextual questions such as how time of day and location affect usage and whether the usage of one

application indicates usage of another [20]. The motivating study done for *SUGILITE*, which explored the repetitive tasks that mobile application users perform, is not publicly available [18]. The *SUGILITE* paper only provides eight tasks performed by their system.

CHAPTER 3: SYSTEM OVERVIEW

The design of Aqueduct is motivated by the observation that keywords in a task description tend to be present in the UI data. Hence, for finding task based application suggestions and entry points, we use design and interaction data captured directly from applications, and to a lesser extent, the application descriptions from Google Play Store.

We define the task-based entry points as connections that allow users to directly navigate to a UI state that is capable of performing the desired task, while also providing the region on the screen that is most relevant to the given task. By providing the most relevant regions within the screen, users will able to quickly gain insights on the suggested applications capabilities.

There are three major subsystems that makes Aqueduct work. First is an Android app UI mining infrastructure that enables capturing design and interaction data while an Android application is being used. This is a web-based infrastructure, and builds on previous work, and has been improved to be more modern, scalable and reliable. The system exposes emulators and devices running a modified version of Android, which can be used to explore apps, during which all screens, gestures, and render-time properties of the components on the screen (i.e., view hierarchy) are recorded. This human-powered app UI mining provides us with the data which enables the entire system.

The second subsystem is a search server, which uses the collected data to find applications and screens which are relevant for a given task. The data is first preprocessed to extract relevant semantic representations and texts from the screens. The natural language representation of tasks are then matched to screens using this extracted data, and the resulting screens are used to provide app and app entry point suggestions for a given query.

Finally, a user facing Android application exposes a multi-modal UI for users to enter their query, and browse the results returned by the server. The results showcase the apps capable of performing the given query, and a screenshot of the entry point of the task for each app. If an app is installed, the user can navigate to the entry point locally using an extensible framework which can perform additional actions for task automation such as automation of touch and text inputs.

Each subsystem is explored in detail in the following chapters, and an overview can be found in Figure 3.1.

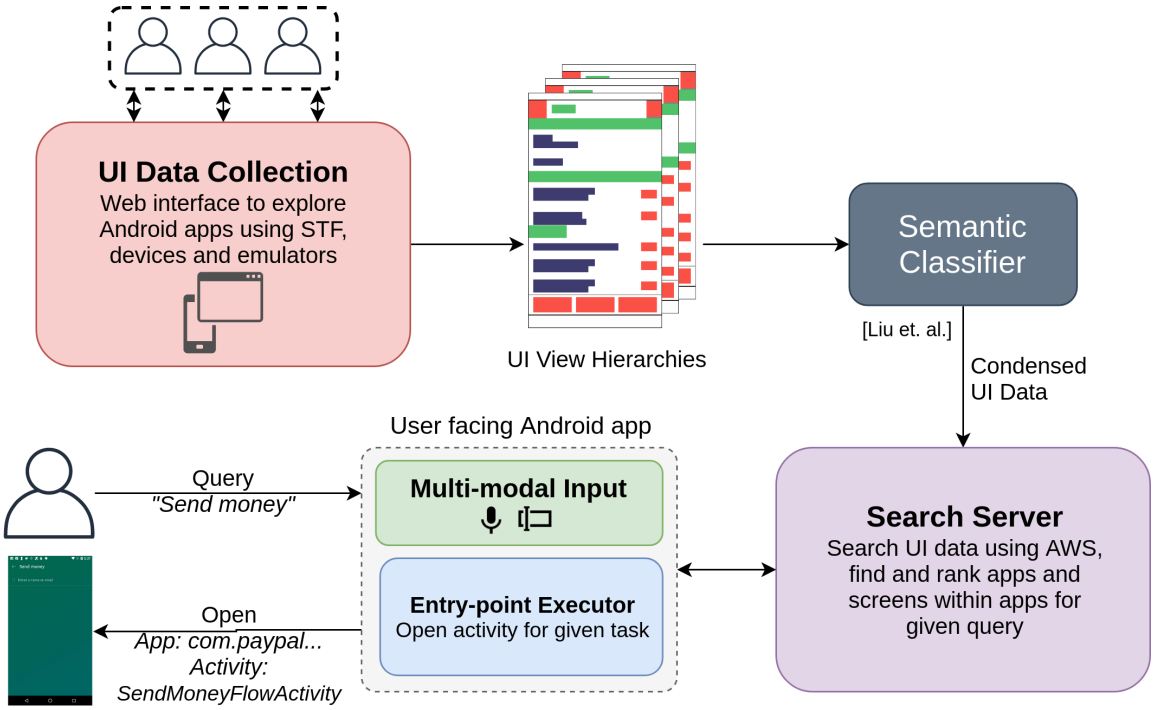


Figure 3.1: Aqueduct uses app UI data mined using a web-based infrastructure. A search server extracts and indexes relevant UI data, and searches over this data given a query. A user facing Android app accepts queries, and opens a matching app at a relevant entry-point.

CHAPTER 4: APPLICATION UI MINING

Android application UI mining is done through a web-based infrastructure, which builds upon the work done in *Rico* [4]. This infrastructure allows capturing of UI data while a user explores an application, including screenshots for each UI state within the application, along with the render-time properties of the contained elements (i.e., view hierarchy), and the user interactions performed on a UI state resulting in transitions to other UI states within the application (i.e., gestures). We improved the system to make it more scalable and reliable, while simultaneously upgrading the components to the latest versions.

The mining approach used here is primarily human-powered, because human explorations tend to be more realistic, and deal with blocking UI states such as log in or sign up screens in a much more efficient way compared to automated UI exploration tools [25].

As shown in Figure 4.1, this setup consists of a pool of Android devices and emulators which are set up at a server. Users can interact with the devices through a web-interface, which is made possible by a modified version of stf, and open source remote access web application [26]. An admin panel is also provided for controlling application installs and access to devices.

4.1 BUILDING ANDROID FOR DATA COLLECTION

4.1.1 Collecting required data points

The first step in application UI mining is to provide a pool of devices which can be used to run Android applications. Out of the three data points that we collect, two - screenshots and gesture data - can be obtained without any modification to the Android system, since the devices are connected to the server using android debug bridge (adb). However, view hierarchy is not exposed directly by the Android system. View hierarchy is a hierarchial representation of UI elements that is present on screen. It contains static and render-time information about every element (`View` in Android terminology) displayed on screen, such as text inside a view, class name, visibility, bounds of the view and so on. This can include sensitive and private user information as well, and hence it's no surprise that this information is not exposed to the outside world.

However, the view hierarchy data is essential to find screens that match a given task, and to facilitate that, we build our own version of Android from Android Open Source Project (AOSP) [27]. We started with the source of the latest stable Android version at the time of

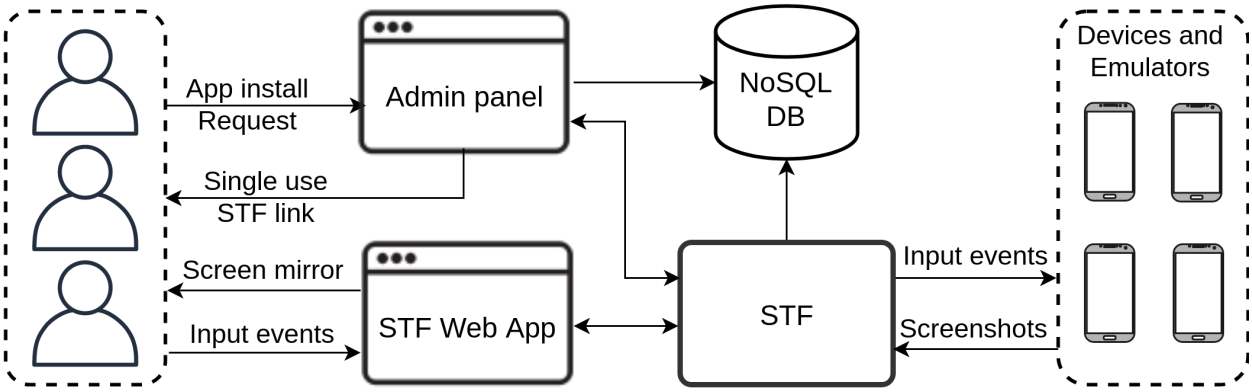


Figure 4.1: Architecture of the application UI data collection architecture. Devices and emulators running a modified version of Android can be remotely accessed to explore apps, and the view hierarchies, screenshots and user inputs are saved during this exploration.

writing (Android 9, codenamed Pie), and added a "view bridge server", which listens to a local socket for view hierarchy requests and responds with the current view hierarchy in json format. It builds the view hierarchy by residing in the activity manager, getting the root view of the current screen, and recursively asking view information about the view's children. The current application's package name is also included in the dump. To reduce security risks, text within password fields are scrubbed from the collected data. An abbreviated view hierarchy sample is given below.

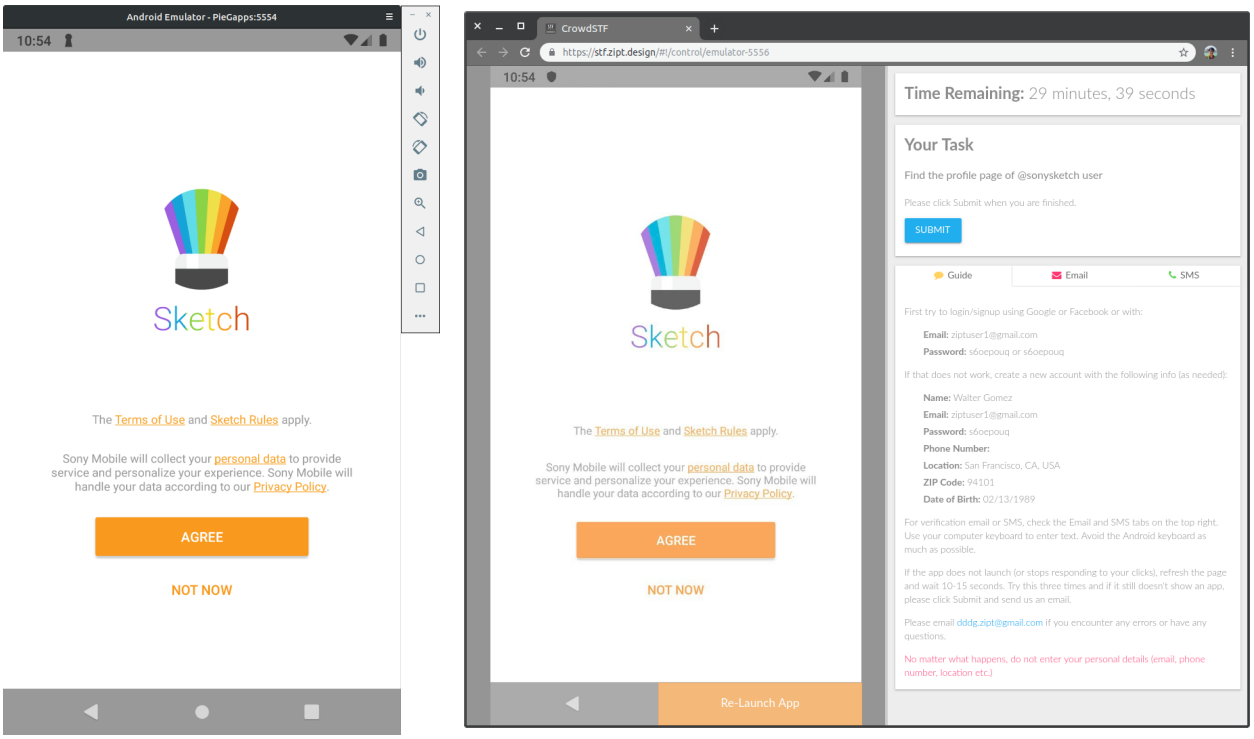
```
{
  "activity_name": "com.company.product",
  "root": {
    "ancestors": [
      "android.widget.FrameLayout",
      ...
    ],
    "class": "com.android.internal.policy.PhoneWindow$DecorView",
    "resource-id": "...",
    "bounds": [0, 0, 1440, 2392],
    "clickable": false,
    "visibility": "visible",
    ...
    "children": [
```

```
      {
        "ancestors": [...],
        "text": "",
        "hint": "...",
        "bounds": [0, 84, 1440, 252],
        "clickable": true,
        "class": "org.cocos2dx.lib.Cocos2dxEditBox",
        "resource-id": "...",
        "visibility": "visible",
        ...
        "children": [...],
      },
      ...
    ]
  }
}
```


4.1.2 Using emulators for data collection

Using physical Android devices limits the scalability of this infrastructure, since the number of available devices determine the number of users that can simultaneously explore applications. To overcome this, we extended the system to work with Android emulators. Android emulator uses qemu to run virtual android devices, and is mostly intended to be used for testing Android applications [28]. Emulators provide almost all capabilities of a real Android device, and hence is a good platform to run Android apps for data collection. The capabilities that can be simulated include location (GPS data), phone calls and messages, network capabilities, fingerprint sensors and other hardware sensors. Performance and UI responsiveness of the emulator is also comparable to real devices if we use Android built for `x86_64`, since it can take advantage of hardware acceleration from the host machine. Additionally emulators allows creating devices with varying screen sizes and resolutions, which could be useful for collecting app responsiveness data.

However, there are a few disadvantages to using the emulator for data collection as well. Since a majority of real Android devices run on ARM processors, a few applications contain prebuilt libraries only for ARM, and does not support `x86` architecture. Running ARM based emulators is possible – but this is not very performant. Another downside is that emulator lacks Bluetooth support, and a few applications require Bluetooth hardware to



(a) (b)

Figure 4.2: (a) Emulator running on the data collection server. (b) Web interface which uses STF to remotely access the emulator.

be present to function. To overcome these problems, the system supports real devices and emulators simultaneously. The few apps which cannot run on emulators will be installed on real devices connected to the server.

To create Android virtual devices for data collection, we built `x86_64` system images for Android emulator from the modified AOSP codebase. Additionally, we automated the process of creating an Android virtual device from the built system images through a python script. We can create and deploy as many emulators as constrained only by CPU and memory resources available in the host machine. However, STF does not directly support emulators, and the modifications done to add this support are described in the next section.

Many Android applications rely on Google Services to be present on a device to fully function, or to function at all. Hence the essential Google services were also incorporated into our AOSP build through Open GApps, for both real and emulated devices [29].

4.2 REMOTE ACCESS WEB INTERFACE

The web interface allows users to control and access the devices used for data collection (Figure 4.2). Using a web app has two main advantages. First, it allows the devices to be controlled from anywhere with an internet connection. Second, it allows the data capture and post-processing to happen in a vastly more powerful server instead of a mobile device. The data collection web interface has two main components – an admin panel which is used to manage app installs and control access to devices, and Smartphone Test Farm (STF), which is an open source Android remote access software [26].

We use STF to allow viewing and controlling a pool of devices remotely through a web browser. STF communicates with the devices using adb, and when a device is in use, captures screenshots from it multiple times a second using `minicap` library [30]. STF also runs a web server which serves a web app which can mirror the screen of any of those devices. The web app establishes a WebSocket connection to the server, and receives screen updates through it. In addition it also captures user interaction data on the mirrored screen, which are sent back to the server through another WebSocket connection. These gestures are converted to Android's user input events such as `TouchDown`, `TouchMove` and `TouchUp`, and sent to the device using the `minitouch` library [31]. Even multi-touch input events can be captured and relayed to the device using this setup.

Changes were made in STF to facilitate UI mining, including storing screenshots captured by `minicap`, user input events captured by the web app, and current view hierarchy obtained from the devices. A token based authentication system was also added to support generating single-use links that can be used to control a particular device. To reduce misuse of devices, these tokens expire 30 minutes from the time a user starts using the device. STF by default expects users to upload Android Package (APK) files (which are essentially installers for Android apps), but to make the experience more seamless, a module to download APK files directly from Play Store was also added to STF. To make this system work with emulators and real devices simultaneously, whenever an application needs to be installed, we download APK files compatible with `ARM64`, and `x86_64` if available.

Finally, the admin panel allows users to submit app install requests, and view recordings in progress, and traces recorded previously. The system contains a polling service which periodically checks for pending app install requests, and requests STF to install apps based on APK availability and device compatibility. Once an app is installed onto a particular device, it retrieves a token from STF, which is then used to create a single-use link to remotely control that device. Clicking on the link directs user to stf to allows remote access this remote access, and starts recording multiple streams of data from that device.

Once the user completes a recording session, a post-processor combines the three data streams – view hierarchies, screenshots, and user input events – into a unified representation called an *interaction trace*. The post-processor also cleans up the data in a few ways. It eliminates duplicate screens without any user events between them, to make the trace compact. In addition, it checks for unwanted elements in the view hierarchies like advertisements, and removes those views. The final trace is then stored on to the database and made available for consumption.

CHAPTER 5: DISCOVERING TASK-BASED ENTRY POINTS

A search server is responsible for discovering applications relevant to the given task, and entry points for that task within those applications. This system processes this UI data, computing screen representations from the render-time properties of the contained elements for each screen. This allows the system to query these screen representations with a given task description and get ranked screens matching the query. The search server then aggregates these screen representations by application and performs a secondary search over the metadata for each application to achieve a ranked list of applications. Finally, it uses a set of heuristics to incorporate the data required for generating deep application links into each application result. Figure 5.1 presents an overview of the search server.

The search server is composed of three major subsystems. The *UI Screen Analyzer* is responsible for preprocessing the application UI data set to compute screen representations for task search. The *Application Suggester* accepts search queries, finds relevant applications and screens within them that match a given query. Finally, the *Entry Point Generator* augments these results with programmatic references to the matched screens, along with additional metadata for each screen. These components are described in detail in the following sections.

5.1 UI SCREEN ANALYZER

This subsystem computes screen representations from the UI data by using pertinent information that is likely to help match task descriptions (e.g. "order a coffee"). The UI data is collected using the system described in the previous section, and is further augmented with semantic annotations by following the methodology described by Liu et al., which produces structural and functional roles for every UI element present within the view hierarchy [5].

Having these semantic annotations enables Aqueduct to use information about icons and text buttons on a screen. These components are likely to match the main action or the verb in the query (e.g. "order"). On the other hand, the object of the query (e.g. "coffee") is more likely to be present in text fields within a screen. As a result, we compute a screen representation that contains semantics of icons (e.g. Settings Gear, Date Range, More Dots), button text (e.g. Submit, Finish, Checkout), and the text fields.

Some of the text on the screens lead to false matches with the search query occasionally, in cases such as "check email", as nearly every application mentions email in multiple locations due to login, etc., despite not being an email application. We found that including Google

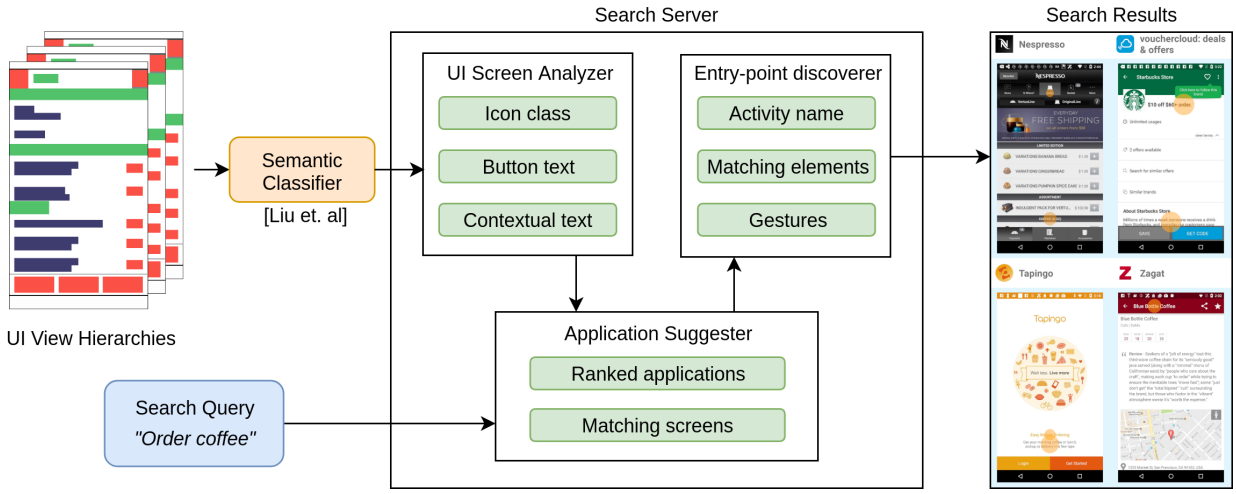


Figure 5.1: An overview of Aqueduct's search server. It takes in a query ("order a coffee") and searches over semantic information from application UI data, and returns ordered search results including applications that can fulfill the query, relevant screenshots, and entry points into the applications that can perform the associated action.

Play description of applications in the search data tend to counteract this and provide better results. So for each application, we extract words from application descriptions excluding common stop-words, and attach those to the corresponding screen representations. Here is a representative snippet from a JSON file encoding a screen:

```
{
  "texts": [
    "Watch WDSU News On Demand",
    "NOWCAST",
    ...
  ],
  "buttons": [
    "delete",
    "share"
  ],
  "icons": [
    "play",
    "warning",
    "time",
    ...
  ],
```

```
  "description": [
    "real-time",
    "news",
    "local",
    ...
  ]
}
```

5.2 APPLICATION SUGGESTER

This subsystem takes in a search query and produces application suggestions for that query. To accommodate natural language queries, we first perform a basic stemming of the query, and remove some common stop words that do not add value to the task description. We then match this processed query to screen representations calculated by the *UI Screen Analyzer*using an off-the-shelf cloud search service that provides keyword search, in order to find a set of screens that match the given query. The search also accommodates for synonyms of common actions found in mobile applications, derived from UX concepts [5]. For example, "chat" will match screens containing "message", "comment" and so on.

The search service assigns a score that indicates how well it matches the query, to each screen representation in the result. Screens below a threshold score are discarded, and the remaining ones are then aggregated by application, and each application in the resulting list is assigned an overall score equal to a weighted sum of the individual screen scores.

Additionally, we include a popularity score for each application, which is a product of its average rating and the number of ratings on Google Play Store. Empirical observations showed that this is a simple but efficient representation of an application's popularity. The application results are sorted by popularity by default, but both popularity and search scores are included in the results so that clients can perform advanced sorting if required.

5.3 ENTRY POINT GENERATOR

For applications that have multiple candidate screens, we apply a set of heuristics to determine which screen most likely serves as the entry point for the desired task. We select the screen with the highest score and include the name of that UI component (i.e. Activity name in Android terminology) which can be used to automatically navigate users to this screen if that application is installed locally on the user's device.

Additionally, we locate the elements within each screen that matches the query, get their position on the screen from the view hierarchy, and add it to the results. These allow us to offer screens relevant to the given task in the search results, as well as draw the users' attention to the parts of the screen relevant to the task.

5.4 IMPLEMENTATION

The *UI Screen Analyzer* is implemented as a Python script, which computes screen representations from interaction mining data and divides it into batches. The *Application Suggester* and *Entry Point Generator* are implemented using Node.js, and can be accessed using a REST API[1]. We utilize AWS CloudSearch to perform the search and ranking of screens and applications.

[1]For example, `http://tidal.eaux.design/search/tool?q=chat`

CHAPTER 6: NAVIGATING TO TASK-BASED ENTRY POINTS

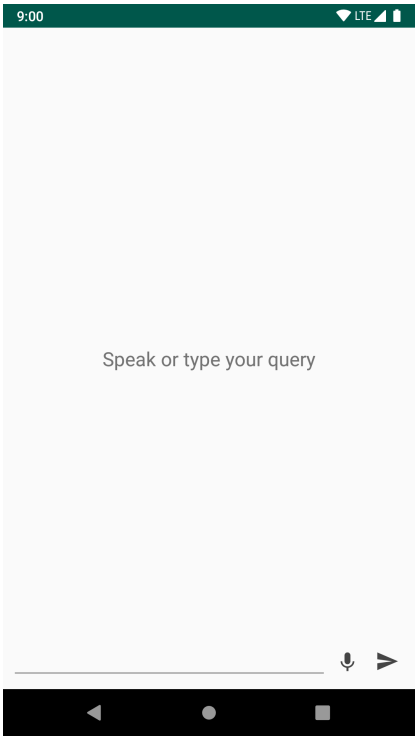
The primary interface to use Aqueduct is built as an Android application. This app serves two purposes. First, it acts as an interface to query the search server and browse the results visually. Users can input their query using text or voice input, and the results are shown as a list that includes application icon, name, and the screenshot of the screen that is recommended by *Entry Point Generator*. Areas of the screen containing text/icons matching the user query are highlighted so that users can quickly determine whether the app is suitable for their query task. An example user flow in the application is shown in Figure 6.1.

The second is an entry point execution system, which can navigate users directly to discovered entry points in locally installed applications. This is built on top of an extensible framework, which can perform additional actions and can be used as a base for task automation systems. These two aspects of the Aqueduct app are explored in detail below.

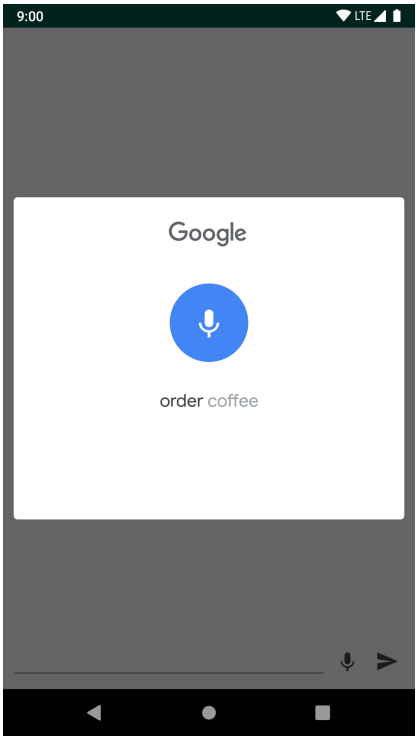
6.1 VISUALIZING SEARCH RESULTS

The home screen of the app presents a minimalist way to accept task queries from the user. Since the system is designed to augment voice assistants, the primary input modal in the system is voice as well. However, there are situations where voice input may not be viable – this has been acknowledged recently by most voice assistants by providing alternate text entry options to the user. Aqueduct follows this pattern and allows users to type in their query in addition to voice input.

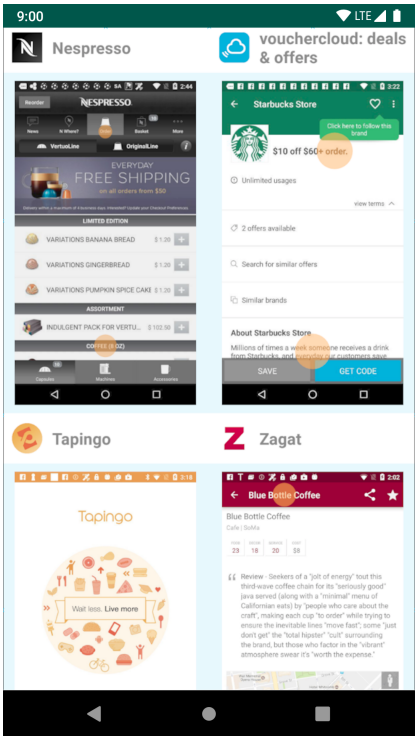
Once the user enters the query, the app sends it to the search server using the exposed REST API, and retrieves the results. These results are shown as a scrolling grid, with each result entry displaying app icon, app name, and screenshot of the task entry point (Figure 6.1c). In addition, the screenshot also highlights parts of the screen, including text and icons, that match the query (the translucent orange circles in Figure 6.1c). These highlights show why the screen is included in the results, in addition to helping users determine whether the app is suitable for the task in hand. The results are not filtered based on whether an app is installed locally on the user's device. Clicking on an installed app results in the entry point execution system taking over, and a non-installed app will result in navigating to the Play Store listing of that app.



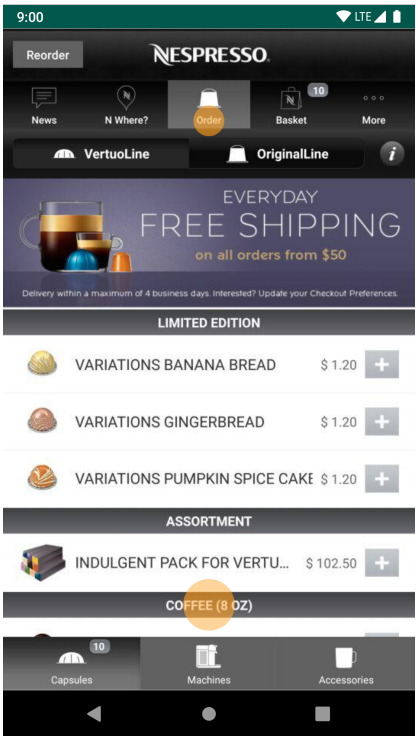
(a) Home screen



(b) Voice input



(c) Query results



(d) Result entry point

Figure 6.1: Screenshots demonstrating how the app assists users in finding task based app-entry points and navigate to them.

6.2 NAVIGATING TO ENTRY POINTS

Aqueduct app contains a task execution framework, in which a task is specified as a list of steps. Each step comprises of an *operation* (e.g., opening a particular application), and an optional *outcome* (e.g., waiting for an app to be in foreground). The execution framework executes each step one after the other, waiting for the outcome of a step before moving on to the next one.

Navigating to a task-based entry point from the search results is defined as a single-step action, where the operation performed is opening the corresponding high-level app UI component (which is identified by `package name` and `activity name` in Android). Starting an application in the state given by this component is done through *intents* in Android, which are essentially requests between application components that contain descriptions of operations to be performed [32]. We construct appropriate *intents* from the relevant activity names, which are then sent to the Android system to open the starting screen of the deep link. Positional data of the matching component is also used in combination with activity names to provide visual hints that show the next steps users can follow within a matched application.

Most activities excluding home screens of applications are usually not 'exported' by app developers, meaning they can usually be opened only by that app itself. This poses a challenge to Aqueduct app, since the entry points discovered are often not exported, and cannot be directly opened by other apps running in user space. To overcome this, Aqueduct app requires `START_ANY_ACTIVITY` permission, which is a protected permission, and granted only to apps installed in the system partition and can run with elevated privileges. Hence, for the execution framework to work, the app needs to be bundled with the Android system images itself, which can then be installed onto devices.

This task execution framework was built as a base to add further features to Aqueduct to make it automate the tasks specified by the user. For example, this framework can be used to navigate to fragments within activities by learning common navigation patterns like tabs and navigation drawers. It could also be used could parse parameters from the search query and automatically insert them into text fields within the application, or press the right button and so on. This may be done using programming by demonstration approaches like SUGILITE [18], or using heuristics on mined interaction data and data points available from the view hierarchy (like input labels and hints on the text fields). A proof-of-concept task automation is built as a demo in Aqueduct app, where it responds to the query 'turn on flashlight' by opening a flashlight application and simulating a touch gesture to flip a switch which activates flash.

Since Aqueduct app is meant to be used as a system application, it makes the task execution framework quite powerful. The ability to inject touch events into any app can be obtained using the `INJECT_EVENTS` permission, and is essential for the flashlight demo. However, an app with such privileges can also be a security risk, and that aspect is explored more in limitations.

We should also note that Aqueduct app by itself is not a replacement for existing voice assistants, as it doesn't directly perform many of the common tasks that are built into most voice assistants, such as checking the weather or setting an alarm. Instead, it is intended as a proof-of-concept app that can act as a bridge between voice assistants and third party applications, by redirecting users to an application that can fulfill their task, when the assistant cannot perform that action by itself.

6.3 IMPLEMENTATION

The Aqueduct app was written in Kotlin language, and built using the standard Gradle build system used for building Android applications. Since the app needs to be installed as a system app for task execution, we first generate an unsigned APK of the app, and then sign it using the same private key used to build our modified version of AOSP [33]. Once signed, the APK file is zipaligned as well. The AOSP makefiles are then modified to include the Aqueduct app as a system app while building system images, and any device or emulator with these generated images installed will have Aqueduct app available. The requirement to be a system app is required only for the task execution framework, and is not necessary for simply browsing the search results.

CHAPTER 7: EVALUATION

We conducted a user study to evaluate whether Aqueduct helps users find appropriate applications to complete a task by providing task specific screenshots. We initially ran a pilot study internally and improved the procedure and questions for the final study based on the feedback. The final study involved users generating common smartphone task descriptions and evaluating the results of Aqueduct for those tasks using a device with Aqueduct app installed.

7.1 STUDY DESIGN AND PROCEDURE

To evaluate Aqueduct we needed a representative set of common tasks that mobile users perform. However, as detailed in related works, no prior studies provide a publicly available list of general-use common smartphone tasks suitable for this study, to the best of my knowledge. Hence we designed the study to have two phases – a task generation phase, and using those tasks to evaluate Aqueduct.

In the first phase, participants performed a generative task exercise where we requested them to list five *active tasks* that they commonly accomplish using a smartphone that go beyond merely consuming information. The *active task* phrasing was included because participants had a tendency to include tasks like 'browse facebook' when asked to simply list common smartphone tasks. Since such apps are app specific and fall outside the scope of the problem that we are attempting to solve, we wanted to avoid such task descriptions as much as possible.

In the second phase participants used Aqueduct app, while following a think-aloud protocol, to search for applications that could be used to complete each task from the first phase. Participants were asked to complete a set of questions after every task. The questions were answered using a seven-point Likert-scale from strongly disagree (one) to strongly agree (seven).

We recruited 24 participants by advertising through a university web programming course mailing list. We compensated each participant with a $10 Amazon gift card. Two researchers conducted each session which approximately lasted 30 minutes each.

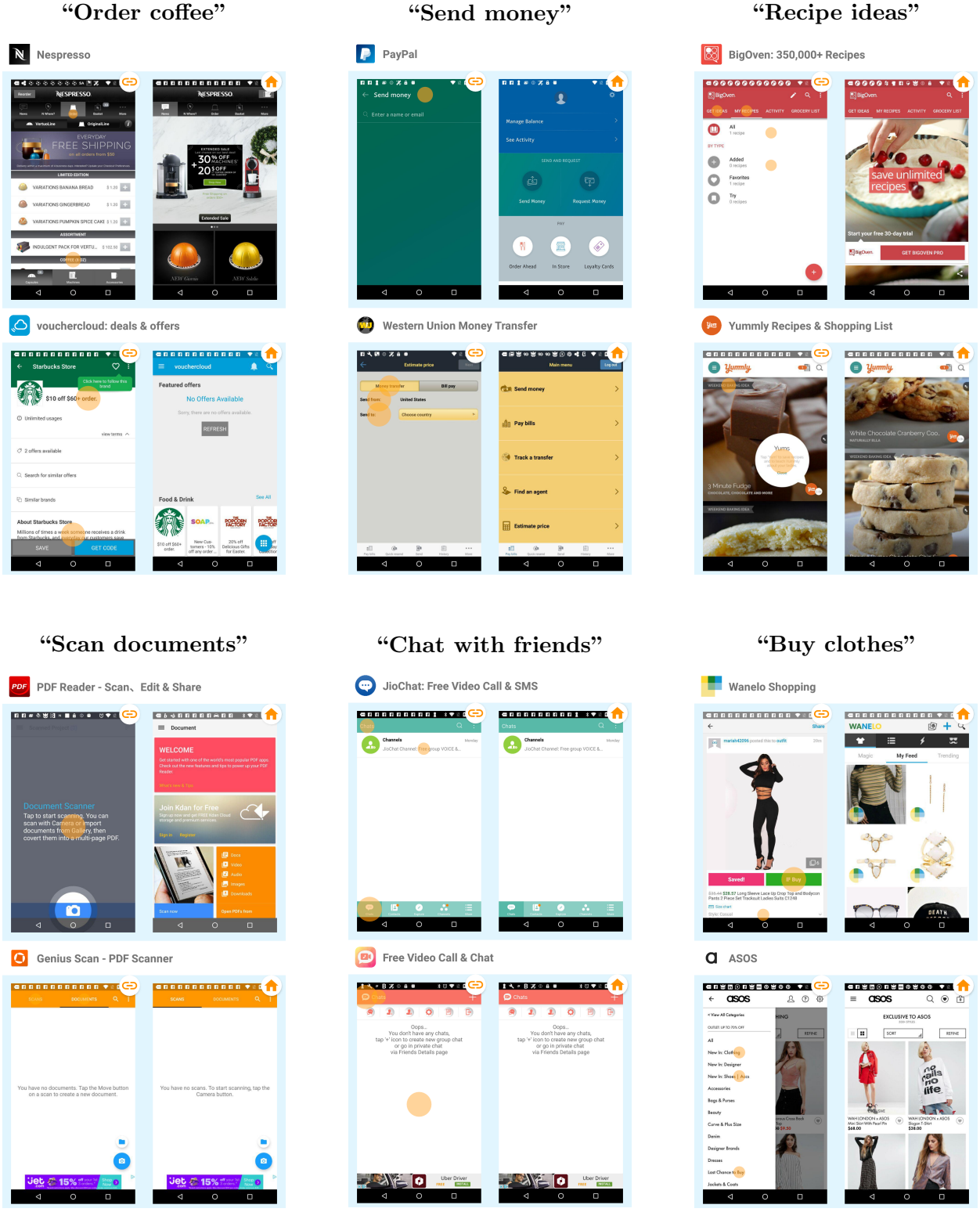


Figure 7.1: Results from Aqueduct for some common queries. Elements matching the query are highlighted for additional clarity. For each application in the results, the matching task-based entry point is shown on the left and the application's home screen is shown on the right.

7.2 RESULTS AND DISCUSSION

Figure 7.1 shows the task-based entry points and home screens of applications returned by common queries to Aqueduct. In a few cases the entry point is the home screen, but we highlight the relevant screen area to focus the user's attention.

7.2.1 Generative Task Phase

In the first phase participants generated 120 tasks. Using a consensus driven, iterative open coding two researchers examined the tasks and identified 23 unique sets of tasks. The tasks generated by users were mostly everyday tasks, such as playing media, communicating with others, taking pictures, and calendar related tasks. The complete set of tasks are listed below. Tasks with exactly the same wording are only included once.

```
watch videos
watch movies
listen to music
browse forums
play music
listen to podcast
identify a song
get sports updates
recipe videos
search online
book a cab
make reservation
set a reminder
set reminder
make payments
make a payment
mobile payments
paying people
send money
check bank account
track spending
deposit a check
message people
message friends
messaging
text friends
communicate with people
chat with friends
video chat
send messages
group chat
check the weather
check weather
what time is it
buy groceries
order food
find deals
find coupons
navigate
navigation
navigate to a destination
get directions
find directions
write email
```

```
respond to emails
check email
set an alarm
set alarm
set alarms
calendar
view calendar
create events
calendar to do list
to do list
taking notes
take notes
add tasks
check calendar
check calendar events
write down notes
view workout activity
track workout
track diet
take photos
take pictures
take a picture
post a picture
post pictures
post updates
turn off lights
translate word
translate words
transit time
calculator
take measurements
show tickets
play games
read books
view document
scan a document
practice flashcards
```

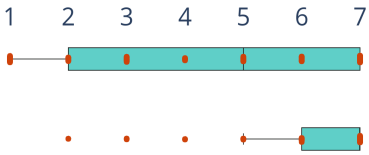
7.2.2 Search Comparison Phase

After every task we had the users answer Q1 and Q2 from Table 7.1. We collected Q2 only when participants found an application matching the task (i.e., a rating of four or more in Q1). Participants were able to find an application for 64% of their tasks, and were satisfied with the discovered entry points in 95% of these cases.

Most participants commented that Aqueduct screenshots were more helpful in determining whether an application in the search results will be useful for a desired task. While looking at results for "communicate with people" P4 exclaimed, "Ooh, this one is clear right off the bat", then proceeded to describe how the screenshots show how to create a group message and share it with someone. Participants also commented that Aqueduct screenshots showed more details about a particular task, compared to app store listings.

The responses to Q1 suggests that Aqueduct is more effective in finding task-based entry points for slightly complex tasks which involved executing multiple steps, such as sending money, navigation and online shopping, as opposed to passive consumption of content, communication, and information lookup (e.g., "watch videos", "chat with friends", "check the

Table 7.1: Questions asked on a seven-point Likert scale and their responses in aggregate.

Question	Results
	
Q1. I was able to find an application that appeared capable of accomplishing the task.	
Q2. It was clear how I would accomplish the task within the app.	

weather").

Q1's responses weren't in favor of Aqueduct in some cases. While Aqueduct's top results were usually relevant, the list sometimes contained irrelevant results further down, and that could be the reason behind this. Almost all participants remarked that augmenting app store search with Aqueduct's algorithm and screenshots would improve application discovery experience.

CHAPTER 8: LIMITATIONS AND FUTURE WORK

8.1 APP UI MINING IMPROVEMENTS

The major limitation of Aqueduct is that it requires mining of application screens and view hierarchies for all applications in its index. Manual exploration of applications to extract this data can be quite labor intensive, given the number of new and updated applications released each day. Automated exploration of applications can overcome this limitation, though the majority of such existing tools focus on UI testing. An application explorer augmented by UI semantics can greatly help automate application design and interaction mining.

Even so, maintaining such a design repository is beneficial for various reasons. In addition to task-based entry point discovery, additional use cases of this data include finding design inspiration, discovering app usability issues, and more [4] [5] [34].

There are further improvements that can be done to make the data collection more scalable and sustainable. Expanding the infrastructure to run emulators in a cloud based service like AWS EC2 can lead to the system having as many devices as required at any given time. We could also provide incentives to application developers to explore their apps using our infrastructure, like analyzing and reporting usability issues.

Currently, our infrastructure only supports data collection from Android applications, and extending it to iOS can not only benefit Aqueduct, but also expand the data-set for comparative studies between the two platforms. Applications are also evolving beyond smartphones to smart devices like televisions, speakers, and other appliances. Exploring, understanding and mining the non-touchscreen user interfaces in such applications could make it possible to extend Aqueduct beyond smartphones.

Some limitations of this system come from the quality of the data itself. Apart from the obvious limitation arising from the number of applications in the data set, most of the data in *Rico* is around two years old, which results in a dated appearance for many screenshots, since mobile application design has evolved rapidly in recent years. Traces of certain applications are incomplete (for instance many of them contain only the log in or sign up screens), and some applications are no longer available for download from Play Store.

8.2 LIMITATIONS ON DISCOVERABLE TASKS

An application may have features that may not be discoverable through interaction mining. Some utilities that provide services like notification mirroring, access to paid services, VR

applications and so on are difficult to mine this way. Users may also have considerations about other aspects of an application before they install it, like whether their friends have presence in a social media or messaging application.

The data collection infrastructure provides fake accounts for users to use, and includes a warning not to enter any personal data while exploring apps. This is essential since the recorded data can be queried and viewed by anyone. However, this also poses some challenges in certain apps, especially banking apps, where fake accounts can't be created. A possible workaround would be to allow users to use their actual accounts, and provide a redaction interface where users can scrub personal information from the recorded data.

8.3 HANDLING COMPLEX TASK DESCRIPTIONS

Aqueduct sometimes struggles with more complex natural language phrasing. More advanced query processing like including a wider range of synonyms in the search will almost certainly lead to better results overall. A related issue is that sometimes text on a screen matches the query even when the application is not applicable to the query. "Check email", in particular, triggered this behavior to a significant extent, since most applications have a log in or sign up screen that mentions email and "check" in field validation error messages.

Processing parameters in the natural language query and handling them gracefully is another avenue for improvement. For example, the user might be inclined to query for "Read Harry Potter" instead of "Read a book", and handling such cases will drastically improve the usefulness of Aqueduct. We can then augment Aqueduct app to understand the UI components of the screen that accepts inputs, and inject the parameters from the user query into these controls automatically.

Another issue is that Aqueduct app sometimes needs to invoke intents that launch activities which are not exported by the app developer, and this may cause crashes, since that activity may be expecting parameters from the invoker, which Aqueduct is unaware of. Since we don't have to worry about this issue with exported activities, a work around would be to identify the previous exported activity from the interaction trace, and navigate to the correct activity by simulating gestures.

This automatic navigation using gestures can also improve cases where apps utilize a single activity containing many fragments. For example, having a primary activity where each tab is a fragment that can accomplish a particular task is a common pattern in Android applications. However, if the UI of the target application gets updated, this method would fail to function.

8.4 GENERATING DEEP-LINK SUGGESTIONS

This would involve two related processes. First, we can expand the list of *actions* supported by the platform itself, using common smartphone tasks. Android has a limited set of built-in actions that currently pertain to food ordering, fitness, finance and ride sharing [35]. We can then use tool to search for applications that support each of those supported tasks to generate reports for app developers informing that they can add support for these actions in their apps. It may even be possible to auto-generate sample code snippets which can be used to add voice assistant integration.

8.5 IMPROVEMENTS IN VISUALIZING RESULTS

Generating dynamic animations or videos on how the application can accomplish a task is also another idea worth exploring. A short animated GIF or a video could be more effective than a list of screenshots to communicate the feature to the user. We have the ingredients for auto-generating such animations – ordered screenshots, and interaction data for each screen. Including more contextual screens might be required in this case to avoid continuity errors in the animation.

8.6 SECURITY CONCERNS

Activity intents, while extremely useful, can lead to some security concerns. One major concern is that applications can hijack communications intended for other applications. Even with newer features in Android designed to alleviate these concerns, vulnerabilities still exist [36], which needs to be addressed for achieving wider coverage of deep links.

The task execution framework in Aqueduct app, while powerful, is also a security concern, since it has permissions to open arbitrary activities and inject touch events in other apps. This approach means that we need to very careful while generating task steps in Aqueduct. However, the common approach in industry when voice assistants need more control and permissions, is to incorporate suitable APIs in the operating system itself. For example, when Google Assistant introduced the capability to scan the current screen to search for relevant information, APIs were introduced in Android to provide a secure way for trusted apps to access the current screenshot and text on the screen.

REFERENCES

[1] J. Kiseleva, K. Williams, J. Jiang, A. Hassan Awadallah, A. C. Crook, I. Zitouni, and T. Anastasakos, "Understanding user satisfaction with intelligent assistants," in *Proceedings of the 2016 ACM on Conference on Human Information Interaction and Retrieval*. ACM, 2016, pp. 121–130.

[2] "Teens use voice search most, even in bathroom, google's mobile voice study finds," 2014. [Online]. Available: https://www.prnewswire.com/news-releases/teens-use-voice-search-most-even-in-bathroom-googles-mobile-voice-study-finds-279106351.html

[3] "Extend your android app," 2019. [Online]. Available: https://developer.android.com/guide/actions

[4] B. Deka, Z. Huang, C. Franzen, J. Hibschman, D. Afergan, Y. Li, J. Nichols, and R. Kumar, "Rico: A mobile app dataset for building data-driven design applications," in *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology*. ACM, 2017, pp. 845–854.

[5] T. F. Liu, M. Craft, J. Situ, E. Yumer, R. Mech, and R. Kumar, "Learning design semantics for mobile apps," in *Proc. UIST*, 2018, pp. 569–579.

[6] B. Yan and G. Chen, "Appjoy: personalized mobile application discovery," in *Proceedings of the 9th international conference on Mobile systems, applications, and services*. ACM, 2011, pp. 113–126.

[7] A. Karatzoglou, L. Baltrunas, K. Church, and M. Böhmer, "Climbing the app wall: enabling mobile app discovery through context-aware recommendations," in *Proceedings of the 21st ACM international conference on Information and knowledge management*. ACM, 2012, pp. 2527–2530.

[8] C. Bernal-Cardenas, K. Moran, M. Tufano, Z. Liu, L. Nan, Z. Shi, and D. Poshyvanyk, "Guigle: A gui search engine for android apps," *arXiv preprint arXiv:1901.00891*, 2019.

[9] N. Chen, S. C. Hoi, S. Li, and X. Xiao, "Mobile app tagging," in *Proceedings of the Ninth ACM International Conference on Web Search and Data Mining*. ACM, 2016, pp. 63–72.

[10] "Handling android app links," 2019. [Online]. Available: https://developer.android.com/training/app-links

[11] "Custom conversational actions," 2019. [Online]. Available: https://developers.google.com/actions/conversational/overview

[12] T. Azim, O. Riva, and S. Nath, "ulink: Enabling user-defined deep linking to app content," in *Proc. MobiSys*, 2016, pp. 305–318.

[13] Y. Ma, X. Liu, M. Yu, Y. Liu, Q. Mei, and F. Feng, "Mash droid: An approach to mobile-oriented dynamic services discovery and composition by in-app search," in *Proc. ICWS*, 2015, pp. 725–730.

[14] "Ui/application exerciser monkey," 2019. [Online]. Available: https://developer.android.com/studio/test/monkey

[15] K. Mao, M. Harman, and Y. Jia, "Sapienz: Multi-objective automated testing for android applications," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ACM, 2016, pp. 94–105.

[16] A. Rosenfeld, O. Kardashov, and O. Zang, "Automation of android applications functional testing using machine learning activities classification," in *2018 IEEE/ACM 5th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. IEEE, 2018, pp. 122–132.

[17] M. Coronado and C. A. Iglesias, "Task automation services: automation for the masses," *IEEE Internet Computing*, vol. 20, no. 1, pp. 52–58, 2016.

[18] T. J.-J. Li, A. Azaria, and B. A. Myers, "Sugilite: Creating multimodal smartphone automation by demonstration," in *Proc. CHI*, 2017, pp. 6038–6049.

[19] "Google duplex: What it is and what google's doing next," 2019. [Online]. Available: https://www.cnet.com/how-to/google-duplex-what-it-is-and-what-googles-doing-next/

[20] M. Böhmer, B. Hecht, J. Schöning, A. Krüger, and G. Bauer, "Falling asleep with angry birds, facebook and kindle: a large scale study on mobile application usage," in *Proc. MobileHCI*, 2011, pp. 47–56.

[21] T. M. T. Do, J. Blom, and D. Gatica-Perez, "Smartphone usage in the wild: a large-scale analysis of applications and context," in *Proc. ICMI*, 2011, pp. 353–360.

[22] S. Zhao, J. Ramos, J. Tao, Z. Jiang, S. Li, Z. Wu, G. Pan, and A. K. Dey, "Discovering different kinds of smartphone users through their application usage behaviors," in *Proc. UBICOMP*, 2016, pp. 498–509.

[23] H. Li, X. Lu, X. Liu, T. Xie, K. Bian, F. X. Lin, Q. Mei, and F. Feng, "Characterizing smartphone usage patterns from millions of android users," in *Proc. IMC*, 2015, pp. 459–472.

[24] Q. Xu, J. Erman, A. Gerber, Z. Mao, J. Pang, and S. Venkataraman, "Identifying diverse usage behaviors of smartphone apps," in *Proc. SIGCOMM*, 2011, pp. 329–344.

[25] B. Deka, Z. Huang, and R. Kumar, "Erica: Interaction mining mobile apps," in *Proceedings of the 29th Annual Symposium on User Interface Software and Technology*. ACM, 2016, pp. 767–776.

[26] "Smartphone test farm," 2019. [Online]. Available: https://github.com/openstf/stf/

[27] "Android open source project," 2019. [Online]. Available: https://source.android.com/

[28] "Run apps on the android emulator," 2019. [Online]. Available: https://developer.android.com/studio/run/emulator

[29] "Open gapps," 2019. [Online]. Available: https://opengapps.org/

[30] "Minicap," 2019. [Online]. Available: https://github.com/openstf/minicap/

[31] "Minitouch," 2019. [Online]. Available: https://github.com/openstf/minitouch/

[32] "Intent," 2019. [Online]. Available: https://developer.android.com/reference/android/content/Intent

[33] "Signing builds for release," 2019. [Online]. Available: https://source.android.com/devices/tech/ota/sign_builds

[34] B. Deka, Z. Huang, C. Franzen, J. Nichols, Y. Li, and R. Kumar, "Zipt: Zero-integration performance testing of mobile app designs," in *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology*. ACM, 2017, pp. 727–736.

[35] "Implement built-in intents for app actions," 2019. [Online]. Available: https://developers.google.com/actions/appactions/bii-integrations

[36] F. Liu, C. Wang, A. Pico, D. Yao, and G. Wang, "Measuring the insecurity of mobile deep links of android," in *Proc. USENIX*, 2017, pp. 953–969.